Question 1:

EXPLANATION

NOTE: This is a lot like the current implementation of the combination problem. There are several differences.

1) The array needs to be sorted first with A.sort() in order to make sure we move up the list correctly and we don't make entries that are essentially the same, such as [1,3,2] and [1,2,3].

2) Instead of passing continuously passing i in the recursively, i+1 must must be used in order to ensure we are checking the index after the current index. We cannot use the same index twice.

3) Since the output list will be in order and there can be multiple [1,2,2] for example, we remove them using the map and converting them to tuple sets to dedupe, then converting back to a list.

```
def amount_helper(nums, start, result, remainder, combination):

    #Base case
    if(remainder == 0):
        result.append(combination)
        return

    elif( remainder <0):
        return # sum exceeded the target

    for i := start to len(nums)-1:
        combination.append(nums[i])
        amount_helper(nums, i+1, result, remainder-nums[i], combination)
        #backtrack
        combination.pop()

def amount(A, S):

    A.sort()
    result = []
    amount_helper(A, 0, result, S,[])
    result = set(map(tuple,result))
    result = list(map(list,result))
```

TIME COMPLEXITY

This is polynomial time O(n^k)

Question 2:

EXPLANATION:

Take in two lists for hunger level per dog and how many treats of any size that are present. The two lists are sorted in reverse so we we feed the hungriest dog the best biscuit size if the biscuit size satisfies the hunger level, meaning it must be bigger than or equal to dogs hunger. The algorithm must return the number of dogs feed. Another variable is used to increment the position of biscuit_size so it is not used again, I call it bisuits_uded. A single loop is used to iterate through each dog. If a biscuit can feed that dog, then that increments biscuits_used (also used as index for biscuit_size), number of dogs feed, and index of hunger_level. The loop will end if all dogs are iterated through or break if all biscuits are used.

TIME COMPLEXITY:

The function uses the python sort function which of O(nlogn) and a single for loop of O(n). This means we have O(nlogn) + O(mlogm) (which is O(nlogn) but for biscuits) + O(n). This means we have a complexity of O(nlogn)